SYCLcon
2021/04/28–29

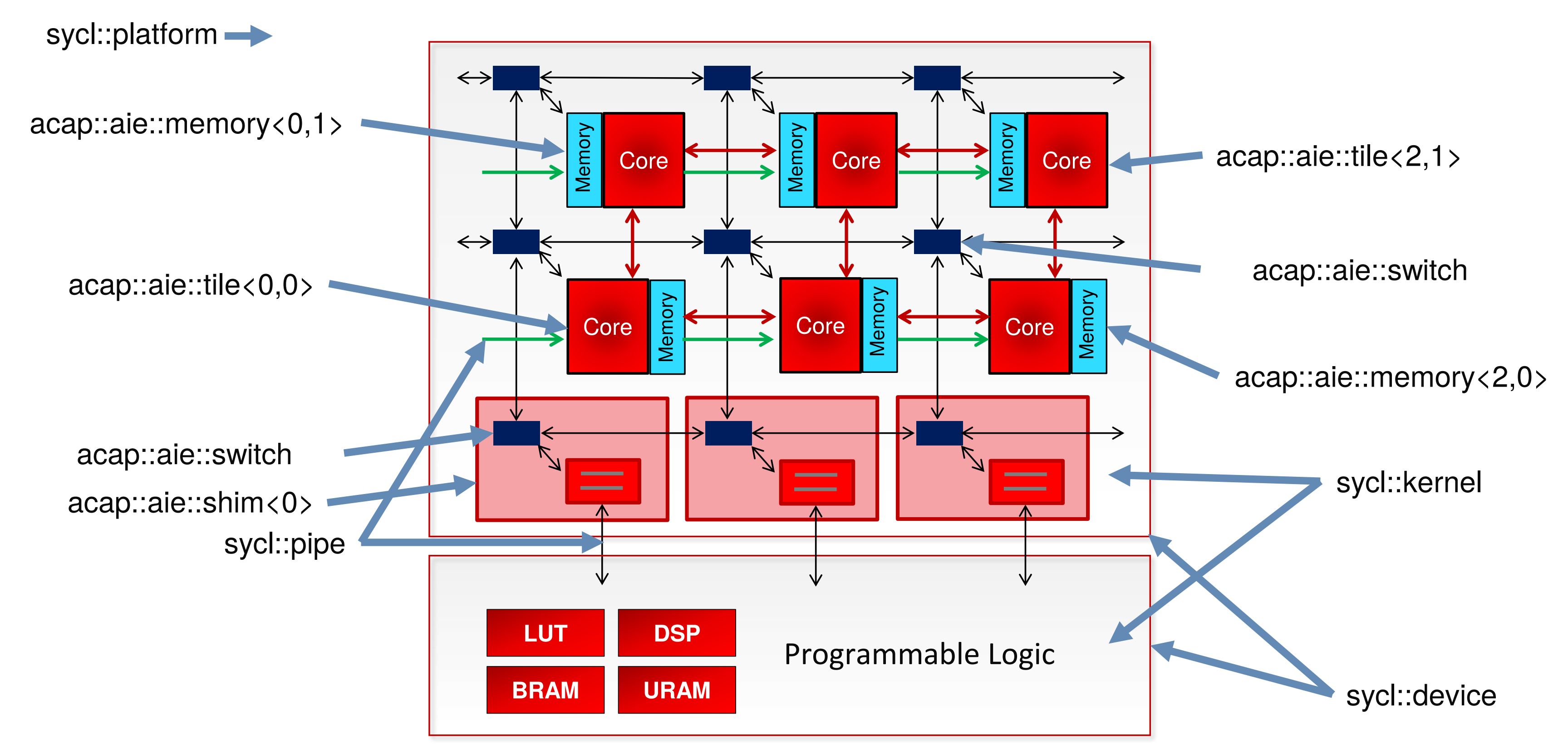
SYCL for Xilinx Versal ACAP AIE CGRA

Andrew Gozillon[2] Gauthier Harnisch[1] Ronan Keryell[1] Hyun Kwon[1]
Ravikumar Chakaravarthy[1] Ralph Wittig[1]
[1]Xilinx [2]University of the West of Scotland

Abstract

SYCL is a single-source C++ DSL targeting a large variety of accelerators in a unified way by using different backends.
Xilinx Versal ACAP is a new system-on-chip (SoC) device integrating various computing resources like various CPUs, an FPGA, a coarse-grain reconfigurable array (CGRA), etc. interconnected by different network-on-chip (NoC).
The AIE CGRA is an array of 400 VLIW DSP operating on 512-bit vectors with their own neighborhood distributed memory (32 KiB data, 16 KiB instructions).
We expose architectural details to the programmer through some SYCL extensions and extend SYCL with a geographical collective model.
The SYCL implementation targeting the AIE CGRA by merging 2 different open-source implementations, Intel's oneAPI DPC++ with some LLVM passes from triSYCL and a new SYCL runtime from triSYCL.
The SYCL device compiler generates LLVM IR for the Synopsys ASIP CHESS compiler generating the AIE instructions.
The host runtime runs on the ARM A72 CPU of the ACAP and controls the CGRA through the Xilinx `libxaiengine-v2` library.

CGRA architecture model and its SYCL model



First approach: expose geographical view of the chip

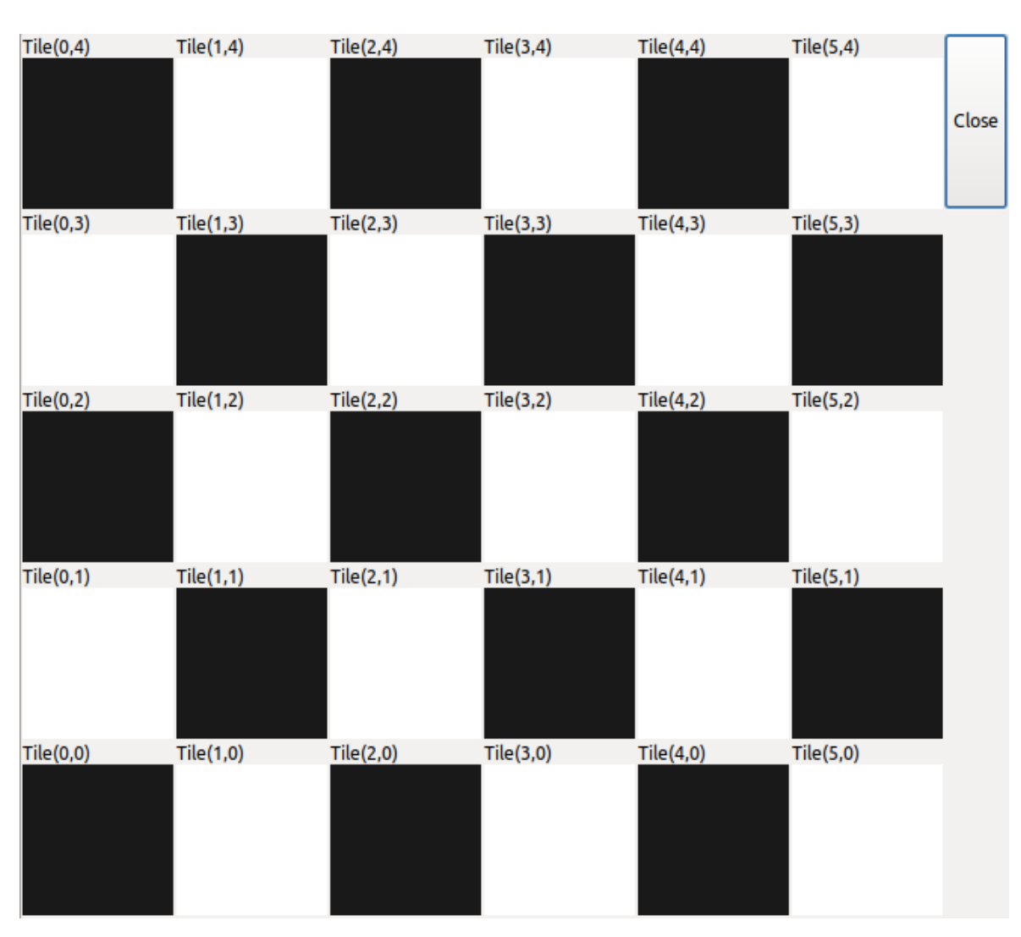
- Each tile is a sub-device with some neighborhood
- Can launch tasks on each tile
- Can use AIE-specific functions inside kernels
  - Neighbor communications
  - DMA
  - Hardware locks
  - ...
- Control of the AIE NoC done by host
- Close to usual SYCL programming model otherwise

SYCL with individual tile model: 1 tile ≡ 1 sub-device

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl::vendor::xilinx;
int main() {
  // Define an AIE CGRA with all the tiles of a VC1902
  acap::aie::device<acap::aie::layout::vc1902> d;
  //  Submit some work on each tile, which is SYCL sub-device
  d.for_each_tile([](auto& t) {
    /* This will instantiate uniformly the same
       lambda for all the tiles so the tile device compiler is executed
       only once, since each tile has the same code
    */
    t.single_task([&](auto& th) {
      std::cout << "Hello, I am the AIE tile (" << th.x() << ',' << th.y()
                << ")" << std::endl;
    });
  });
  // Wait for the end of each tile execution
  d.for_each_tile([](auto& t) { t.wait(); });
  d.tile(3,4).single_task([&] { std::cout << "Hello from (3,4)"
                                          << std::endl; }).wait();
}
```

Second approach: entangled mode by weaving tiles and memories

- Plain SYCL can be cumbersome with 400 tiles to program...
- Provide a cooperative mode allowing meta-programming tiles and memories
  - Compile-time specialization of tiles (instructions) and memories (data)
  - Allow each tile to access neighbor tile memories in a **type-safe** way



Weaving heterogeneous tiles and memories with meta-programming

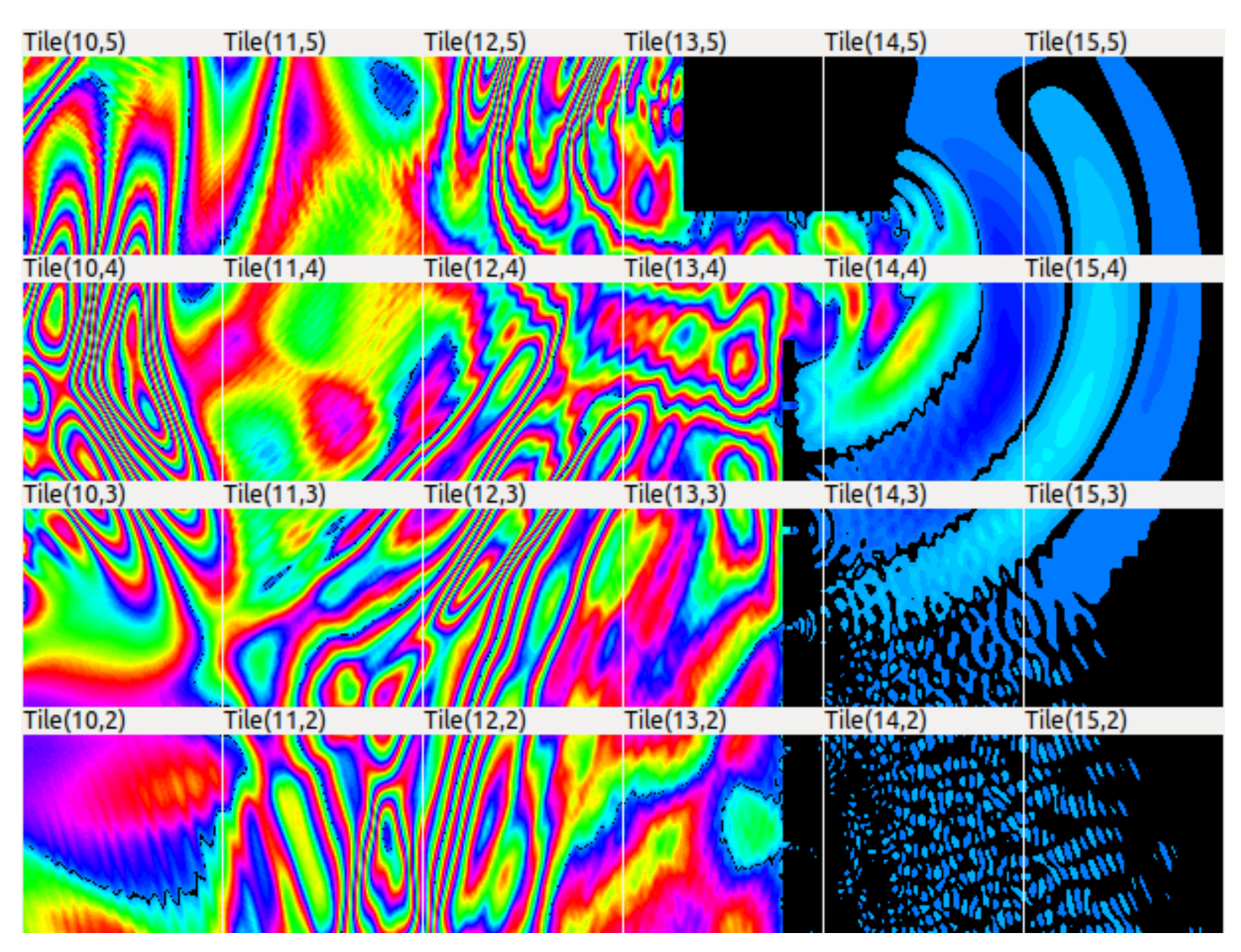
```cpp
#include <sycl/sycl.hpp>
#include <iostream>
#include <type_traits>
using namespace sycl::vendor::xilinx;
// A little graphics debugging environment
graphics::application a;
// To have a checkerboard-like pattern depending on x & y tile coordinate
bool constexpr white_x_y(int x, int y) { return (x + y) & 1; };
// The memory content of a black tile
struct black {
  double d = 5.2;
  static bool constexpr is_white = false;
 };
// The memory content of a white tile
struct white {
  int i = 7;
  static bool constexpr is_white = true;
};
// A memory tile has to inherit from acap::aie::memory<AIE, X, Y>.
// Inherit here from either white or black according to white_x_y(X, Y)
template <typename AIE, int X, int Y>
struct tile_memory : acap::aie::memory<AIE, X, Y>
                   , std::conditional_t<white_x_y(X, Y), white, black> {};

// A program tile has to inherit from acap::aie::tile<AIE, X, Y>
template <typename AIE, int X, int Y>
struct tile_prog : acap::aie::tile<AIE, X, Y> {
  using t = acap::aie::tile<AIE, X, Y>;
  // Tile code
  void operator()() {
    auto &own = t::mem();
    if constexpr (t::mem_t::is_white) {
      std::cout << " i = " << own.i << std::endl;
      a.update_tile_data_image(t::x, t::y, &own.i, 5, 7);
    }
    else {
      std::cout << " d = " << own.d << std::endl;
      a.update_tile_data_image(t::x, t::y, &own.d, 5, 7);
    }
  }
};

int main(int argc, char *argv[]) {
  // Create a SYCL ACAP device with some AIE tiles
  acap::aie::device<acap::aie::layout::full> aie;
  // Start the graphics display
  a.start(argc, argv, decltype(aie)::geo::x_size,
          decltype(aie)::geo::y_size, 1, 1, 100);
  // Run collective program on the tiles, weaving tile_prog & tile_memory
  aie.run<tile_prog, tile_memory>();
}
```

Type-safe access to heterogeneous neighbor memories (PDE, stencil...)

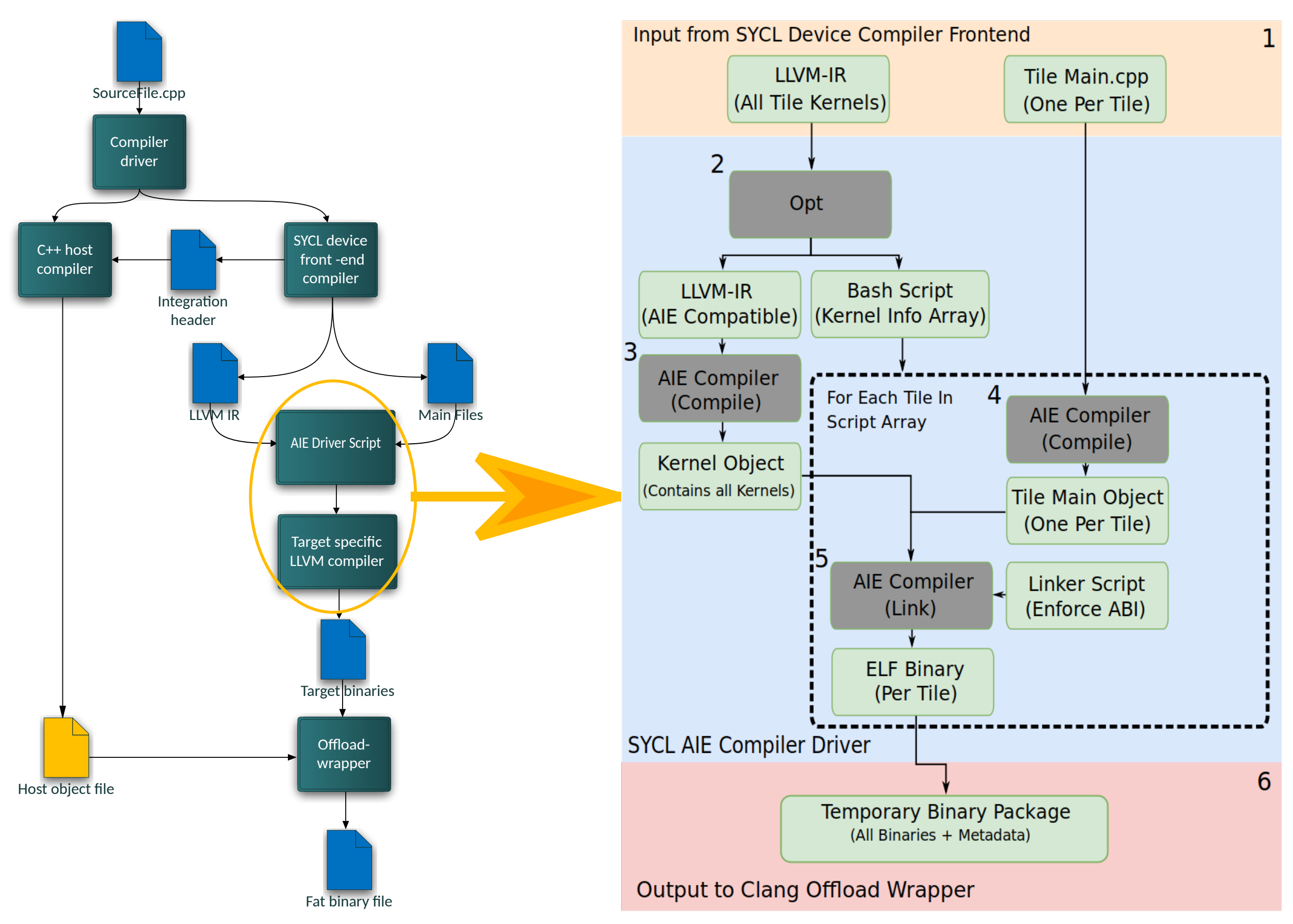
```cpp
auto& own = t::mem();
for (int j = 0; j < image_size; ++j)
  for (int i = 0; i < image_size - 1; ++i) {
    // Compute dw/dx
    auto north = own.w[j][i + 1] - own.w[j][i];
    // Integrate horizontal speed
    own.u[j][i] += north*alpha;
  }
t::barrier();
if constexpr (t::is_memory_module_south()) {
  auto& below = t::mem_south();
  for (int i = 0; i < image_size; ++i)
    below.v[image_size - 1][i] = own.v[0][i];
}
```



Different multi-level implementation/emulation for hardware-software co-design

- Full SYCL compiler & runtime implementation
  - Run on real hardware or hardware simulator
- Pure SYCL C++ implementation
  - No specific compiler required!
  - Run on (laptop) host CPU at full C++ speed, standard debugging, thread-sanitizer of hardware features...
    - 1 thread per host... thread, 1 thread per AIE tile, 1 thread per GPU work-item, 1 thread per FPGA work-item
  - Easy code instrumentation for statistics by adapting SYCL C++ classes
  - Use normal debugger
    - Gdb is scriptable in Python to expose new features
  - Can experiment with Xilinx devices from year 2030
- Mix-and-match
  - Run some parts of the hardware remotely or in simulators
  - Allow kernels on host CPU while using memory-mapped real hardware (DMA, AXI streams, NoC...)
  - Distribute execution across data-center (Celerity SYCL for MPI+SYCL)

Implementation


SourceFile.cpp
Compiler driver
C++ host compiler
Integration header
SYCL device front -end compiler
LLVM IR
Main Files
AIE Driver Script
Target specific LLVM compiler
Target binaries
Host object file
Offload-wrapper
Fat binary file
Input from SYCL Device Compiler Frontend
1
LLVM-IR (All Tile Kernels)
Tile Main.cpp (One Per Tile)
2
Opt
LLVM-IR (AIE Compatible)
Bash Script (Kernel Info Array)
3
AIE Compiler (Compile)
Kernel Object (Contains all Kernels)
For Each Tile In Script Array
4
AIE Compiler (Compile)
Tile Main Object (One Per Tile)
5
AIE Compiler (Link)
Linker Script (Enforce ABI)
ELF Binary (Per Tile)
SYCL AIE Compiler Driver
6
Temporary Binary Package (All Binaries + Metadata)
Output to Clang Offload Wrapper

Conclusion

- Provide direct programming in C/C++ and SYCL for CGRA like Xilinx ACAP VC1902
  - Complement existing Xilinx ML environment and Xilinx ADF for Kahn's Process Network
- Follow SYCL philosophy: based on plain C++ without extensions
- Implementation based on triSYCL, oneAPI DPC++ and Synopsys ASIP Designer `chess-clang`
- Single-source pure C++ is incredibly powerful for hardware-software co-design
  - Provide CPU emulation for architecture research and co-design
  - Can detect hardware concurrency issues with CPU emulation
- To be open-sourced and merged into `https://github.com/triSYCL/triSYCL`
- Retargetable to other CGRA